Instituto Politécnico Nacional

Escuela Superior De Computo

Sistemas en Chip

Práctica 7

Nombre de los integrantes:

García Quiroz Gustavo Iván

Bejarano García Owen Uriel

Grupo: 7CV3

Nombre del Profesor: Miguel Ángel Castillo Martínez

Fecha De Entrega: 03/06/2025

1 Índice

2 Introducción

En esta práctica se implementará un sistema de control y medición de velocidad de un motor DC con encoder, utilizando un Arduino UNO, un driver L298N y tres temporizadores configurados en diferentes modos de operación, el Timer0 se utilizará en modo normal para gestionar funciones básicas de temporización, el Timer1 en modo PWM para el control preciso de velocidad del motor y el Timer2 en modo CTC (Clear Timer on Compare) para generar interrupciones periódicas y medir tiempos de ejecución.

Además, se buscó analizar el comportamiento del motor mediante la lectura del encoder, calculando su velocidad en RPM (Revoluciones Por Minuto) y registrando los tiempos de interrupción y ejecución de tareas. Esta práctica es fundamental para entender el manejo de temporizadores en microcontroladores, así como las técnicas de medición de velocidad en sistemas mecatrónicos. Los resultados obtenidos permiten evaluar la eficiencia del control PWM y la precisión del encoder en aplicaciones de robótica y automatización.

3 Objetivo General

Diseñar y programar un sistema de control de velocidad para un motor DC utilizando temporizadores del microcontrolador ATmega328P (Arduino UNO), aplicando técnicas de modulación por ancho de pulso (PWM) y lectura de encoder incremental para medir su velocidad angular en tiempo real.

3.1 Objetivos Específicos

- Configurar y utilizar los temporizadores Timer0, Timer1 y Timer2 del ATmega328P en diferentes modos (Normal, PWM Phase Correct y CTC) para controlar y medir eventos temporales.
- Generar señales PWM mediante Timer1 para controlar la velocidad de un motor DC a través de un puente H (L298N).
- Implementar interrupciones externas para contar los pulsos generados por el encoder incremental y calcular la velocidad angular (RPM) del motor.
- Visualizar y validar las señales PWM y del encoder utilizando un osciloscopio digital.
- Calibrar el sistema para obtener mediciones de velocidad precisas comparando los resultados del programa con observaciones físicas y señales en el osciloscopio.

4 Materiales y Equipo Utilizado

Componentes Electrónicos

- **Arduino UNO (ATmega328P):** Microcontrolador principal para control de motor, generación de PWM y lectura del encoder.
- **Driver L298N:** Puente H utilizado para controlar la dirección y velocidad del motor DC.
- **Motor DC con reductor y encoder incremental:** Proporciona realimentación de velocidad mediante señales en cuadratura (canales A y B).
- **Osciloscopio digital:** Para visualizar señales PWM y de encoder, validar tiempos y frecuencias.
- **Protoboard y cables Dupont:** Conexión de componentes sin soldadura.
- **Fuente de alimentación externa (opcional):** Para alimentar el motor independientemente del Arduino**.**

Software

- **Arduino IDE:** Entorno de desarrollo utilizado para programar el ATmega328P en C/C++.

5 Marco Teórico

5.1 Temporizadores en Arduino UNO (ATmega328P)

Los temporizadores (timers) son módulos hardware del microcontrolador ATmega328P (usado en Arduino UNO) que permiten medir intervalos de tiempo, generar señales PWM y gestionar interrupciones. En esta práctica se utilizan tres temporizadores con modos de operación distintos:

5.1.1 Timer0 (8-bit) – Modo Normal

- **Función principal:**

Generar una interrupción cada ~1 segundo para mostrar el valor actual de RPM por UART.

- **Configuración en la práctica:**

Modo Normal (WGM0[2:0] = 0b000), donde el timer incrementa desde 0 hasta 255 y dispara una interrupción por overflow (TOV0) y se usa prescaler de 1024.

- **Frecuencia de operación:**

Con prescaler de 1024 y reloj a 16 MHz:

$$f_{Timer0} = \frac{16\,MHz}{1024 \times 256} = 61.035\,Hz \qquad (Periodo \approx 16.38\,ms)$$

Se cuentan 61 overflows (~1 segundo) antes de imprimir las RPM.

5.1.2 Timer1 (16-bit) – Modo CTC (Clear Timer on Compare)

- **Función principal:**

Calcular las RPM del motor a partir de los pulsos del encoder cada 100 ms.

- **Configuración en la práctica:**

Modo CTC (WGM1[3:0] = 0b0100), donde el timer se reinicia al alcanzar el valor en OCR1A = 24999, con un prescaler de 64.

- **Frecuencia de interrupción:**

$$f_{interrupción} = \frac{16MHz}{64 \times (24999 + 1)} = 10Hz \quad (cada\ 100\ ms)$$

En la ISR de TIMER1_COMPA_vect, se calcula la diferencia de pulsos y se convierte a RPM usando:

$$RPM = \frac{\Delta pulsos \times 600}{PULSOS\_POR\_VUELTA}$$

5.1.3 Timer2 (8-bit) – Modo Fast PWM

- **Función principal:**

Generar una señal PWM en el pin PB3 (OC2A) para controlar la velocidad del motor mediante el puente H (INB del L298N).

- **Configuración en la práctica:**

Modo Fast PWM de 8 bits (WGM2[2:0] = 0b011), con salida no inversora (COM2A1:0 = 10) y prescaler de 64.

El ciclo de trabajo (duty cycle) se ajusta escribiendo en el registro OCR2A, con valores recibidos por UART.

- **Frecuencia de PWM:**

Con prescaler de 8:

$$f_{PWM} = \frac{16\ MHz}{64 \times 256} \approx 976.56\ Hz$$

Esto proporciona un PWM lo suficientemente rápido para aplicaciones de control de motores.

5.2 Encoders y Medición de Velocidad

5.2.1 Encoder Incremental

Principio de funcionamiento: Sensor que genera pulsos digitales en función del giro del motor. En esta práctica, se utiliza un solo canal (A) del encoder, que produce una cantidad fija de pulsos por revolución (PPR: Pulsos Por Revolución).

Conteo de pulsos: Cada vez que ocurre un flanco ascendente en la señal A del encoder, se incrementa una variable (pulsos_encoder) mediante una interrupción externa (INT0).

Cálculo de la velocidad angular (RPM): La velocidad se actualiza cada 100 ms en la interrupción del Timer1, calculando la diferencia de pulsos desde la última medición.

Se usa la fórmula antes vista:

$$RPM = \frac{\Delta pulsos \times 600}{PPR}$$

Donde:

- $\Delta pulsos$ es el número de pulsos en los últimos 100ms
- 600 es el factor de conversión para pasar de 100 ms a minutos (60 s/min)/(0.1s)=600
- PPR es el número de pulsos por revolución

5.3 Driver L298N

- **Funcionamiento:** Puente H que permite controlar dirección y velocidad de motores DC.
- **Entradas:**
    - **IN1 e IN2:** Controlan la dirección (HIGH/LOW).
    - **ENA:** Señal PWM para regular velocidad.

- **Conexión con Arduino**

L298N	Arduino
IN1	D4
IN2	D5
ENA	D9 (PWM Timer1)

Tabla 1 Conexión con Arduino

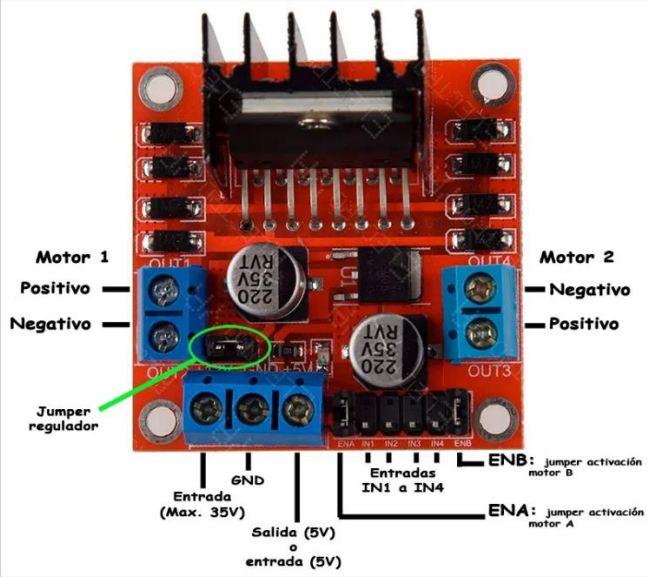


Figura 1 L298N

6 Desarrollo de la Práctica

6.1 Armar el circuito

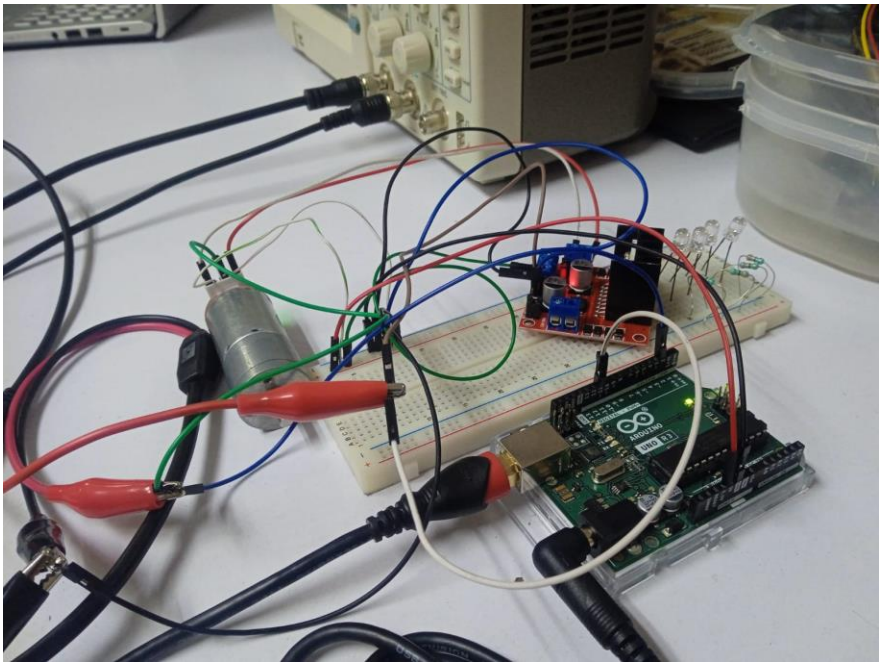


Figura 2. Circuito armado

6.2 Codificación de la práctica

```c
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>
#include <stdlib.h>
#include <util/delay.h>

#define F_CPU 16000000UL
#define BAUD 9600
#define UBRR_VALUE ((F_CPU / (16UL * BAUD)) - 1)

#define PULSOS_POR_VUELTA 11

volatile uint32_t contador_pulsos = 0;
volatile uint32_t rpm = 0;

// // --- Definición de registros (estructuras) ---
typedef struct {
    uint8_t WGMn : 2;
    uint8_t      : 2;
    uint8_t COMnB : 2;
    uint8_t COMnA : 2;
} TCCRnA_t;
```

```
24  typedef struct {
25      uint8_t CSn : 3;
26      uint8_t WGMn : 1;
27      uint8_t      : 2;
28      uint8_t FOCnB : 1;
29      uint8_t FOCnA : 1;
30  } TCCRnB_t;
31
32  typedef struct {
33      uint8_t WGMn : 2;
34      uint8_t      : 2;
35      uint8_t COMnB : 2;
36      uint8_t COMnA : 2;
37  } TCCRnA16_t;
38
39  typedef struct {
40      uint8_t CSn : 3;
41      uint8_t WGMn : 2;
42      uint8_t      : 1;
43      uint8_t ICESn : 1;
44      uint8_t ICNCn : 1;
45  } TCCRnB16_t;
46
```

```
47  // Definimos los punteros para los registros de los temporizadores:
48  volatile TCCRnA_t *TCCR0A_ = (TCCRnA_t *)0x44;
49  volatile TCCRnB_t *TCCR0B_ = (TCCRnB_t *)0x45;
50
51  volatile TCCRnA16_t *TCCR1A_ = (TCCRnA16_t *)0x80;
52  volatile TCCRnB16_t *TCCR1B_ = (TCCRnB16_t *)0x81;
53
54  volatile TCCRnA_t *TCCR2A_ = (TCCRnA_t *)0xB0;
55  volatile TCCRnB_t *TCCR2B_ = (TCCRnB_t *)0xB1;
56
```

```
57
58  // CONFIGURAR ENCODER ==============================
59  void setup_encoder_interrupt() {
60      DDRD &= ~(1 << PD2);     // PD2 (INT0) como entrada
61      PORTD |= (1 << PD2);     // Activar pull-up
62      EICRA |= (1 << ISC00);   // Interrupción en flanco (cambio lógico)
63      EIMSK |= (1 << INT0);    // Habilita INT0
64  }
65
66  // ISR para contar pulsos del encoder
67  ISR(INT0_vect) {
68      contador_pulsos++;
69  }
70
71  //=================================================
```

```
74  void USART_Init(unsigned int ubrr) {
75      UBRR0H = (unsigned char)(ubrr >> 8);
76      UBRR0L = (unsigned char)ubrr;
77
78      UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0); // RX, TX, RX Interrupt
79      UCSR0C = (1 << UCSZ01) | (3 << UCSZ00);               // 8 data bits, 1 stop bit
80  }
81
82  void USART_Transmit(uint8_t data) {
83      while (!(UCSR0A & (1 << UDRE0)));
84      UDR0 = data;
85  }
86
87  void USART_PrintNumber(uint16_t num) {
88      char buffer[10];
89      itoa(num, buffer, 10);  // Convierte el número en una cadena (en base 10)
90      for (int i = 0; buffer[i] != '\0'; i++) {
91          USART_Transmit(buffer[i]);  // Envía cada caracter
92      }
93  }
```

```
void setup_timer0_normal() {
    TCCR0A_->WGMn = 0;  // Normal Mode (WGM01:0 = 00)
    TCCR0B_->WGMn = 0;  // Normal Mode

    TCCR0B_->CSn = 5;   // Prescaler 1024 (CS02:0 = 101)

    TIMSK0 |= (1 << TOIE0); // Habilitar interrupción por overflow
}

ISR(TIMER0_OVF_vect) {
    static uint8_t overflow_count = 0;

    overflow_count++;
    if (overflow_count >= 61) { // ≈ 1s
        overflow_count = 0;

        USART_PrintNumber(rpm);
        USART_Transmit('\n');
    }
}
```

```
void setup_timer1_ctc() {
    TCCR1A_->WGMn = 0;  // Modo CTC (WGM11:0 = 00)
    TCCR1B_->WGMn = 1;  // WGM12 = 1 (CTC), WGM13 = 0
    TCCR1B_->CSn = 3;   // Prescaler 64 (CS12:0 = 011)

    OCR1A = 24999;      // Interrupción cada 100ms

    TIMSK1 |= (1 << OCIE1A); // Habilita interrupción por comparador A
}

ISR(TIMER1_COMPA_vect) {
    static uint32_t last_pulsos = 0;

    uint32_t delta_pulsos = contador_pulsos - last_pulsos;
    last_pulsos = contador_pulsos;

    rpm = (delta_pulsos * 600) / PULSOS_POR_VUELTA; // 600 = 60s * 10 (por 100ms)
}
```

```
// --- Configuración Timer2 PWM ---
void setup_pwm_timer2() {
    DDRB |= (1 << PB3); // PB3 = OC2A como salida (INB del motor)

    TCCR2A_->WGMn = 3;     // Fast PWM (WGM20=1, WGM21=1)
    TCCR2A_->COMnA = 2;    // Clear OC2A on compare match (modo no inversor)
    // TCCR2A_->COMnB = 0;    // No usamos OC2B
    TCCR2B_->WGMn = 0;     // WGM22 = 0 → Fast PWM de 8 bits
    TCCR2B_->CSn = 4;      // Prescaler 64 (CS22:0 = 100)

    OCR2A = 0;             // Duty Cycle inicial = 0% (motor apagado)
}
```

```
148 // --- ISR recepción UART ---
149 ISR(USART_RX_vect) {
150     unsigned char received = UDR0;
151 
152     if (received >= '0' && received <= '9') {
153         uint8_t value = received - '0';  // Convertir ASCII a número
154 
155         if (value == 0) {
156             OCR2A = 0; // Motor apagado
157         } else {
158             // Duty mínimo 30%, máximo 100%
159             OCR2A = ((value - 1) * (255 - 77) / 8) + 77; // 77 ≈ 30% duty
160         }
161     }
162 }
```

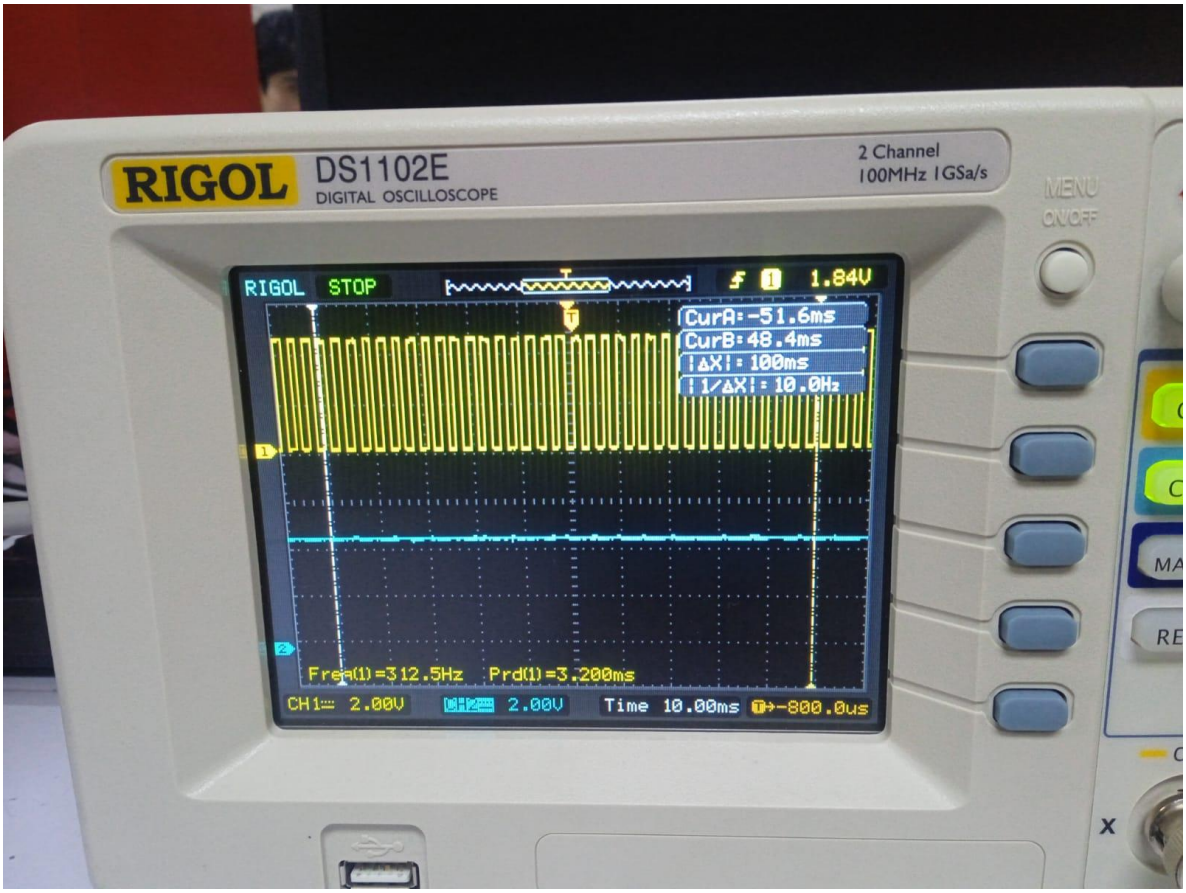
```
164 int main(void) {
165     cli();
166 
167     USART_Init(UBRR_VALUE);
168     setup_pwm_timer2();
169     setup_timer0_normal();
170     setup_timer1_ctc();
171     setup_encoder_interrupt();
172 
173     sei();
174 
175     while (1) {
176 
177         // PWM desde UART ISR
178         // Pulsos contados en INT0 ISR
179         // RPM calculada en Timer1 ISR
180     }
181 }
```

6.3 Medición de Velocidad (Encoder + Interrupciones) y control de Velocidad (10 Niveles PWM)

El encoder incremental genera 11 pulsos por revolución (PPR). Se usó una interrupción con cualquier cambio lógico del canal A del encoder para contar pulsos. La velocidad (RPM) se calculó con la fórmula:

$$RPM = \frac{Pulsos}{PPR} \times \frac{60}{\Delta t(segundos)}$$

Además se definieron 10 velocidades diferentes mediante PWM (0, 77, 99, 121, 143, 165, 187, 209, 231, 255) // 30% a 90%



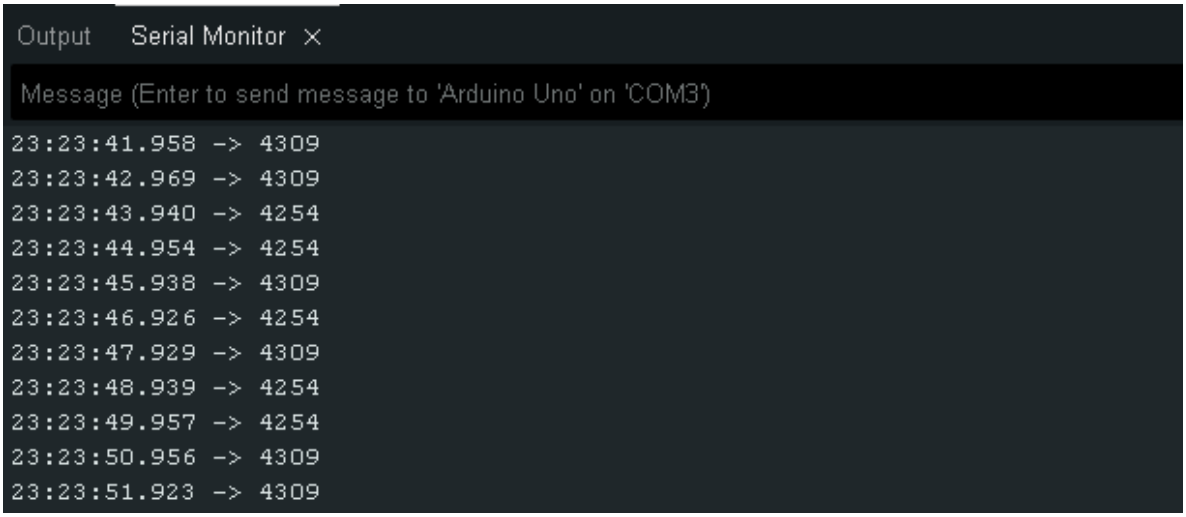


Figura 3. Salida de la terminal y osciloscopio (Encoder y PWM) con un duty cycle de 255 y un intervalo de tiempo de 100 ms

En esta práctica se utiliza un encoder incremental con una resolución de 11 pulsos por revolución (PPR). Se mide la velocidad del motor en revoluciones por minuto (RPM), con valores típicos en el rango de 4200 a 4300 RPM.

Para determinar la cantidad esperada de pulsos que debe generar el encoder en un intervalo de tiempo de 100 ms, se realiza el siguiente cálculo:

1. Convertir la velocidad de RPM a revoluciones por segundo (RPS):

$$RPS = \frac{RPM}{60} = \frac{4200}{60} = 70\ revoluciones\ por\ segundo$$

2. Calcular las vueltas realizadas en 100 ms (0.1 segundos):

$$Vueltas_{100ms} = RPS \times 0.1 = 70 \times 0.1 = 7\ vueltas$$

3. Multiplicar el número de vueltas por los pulsos por revolución para obtener los pulsos esperados:

$$Pulsos_{100ms} = Vueltas_{100ms} \times PPR = 7 \times 11 = 77\ pulsos$$

7 Resultados

7.1 Mediciones de Velocidad vs. PWM

Se registraron las RPM para cada uno de los 10 niveles de PWM:

Duty Cycle (%)	RPM Medidos	Observaciones
0% (0/255)	0	Motor detenido
10% (77/255)	1100-1200 RPM	Velocidad mínima
30% (121/255)	2200-2300 RPM	
50% (165/255)	2900-3000 RPM	Velocidad media
70% (209/255)	3500-3600 RPM	

Duty Cycle (%)	RPM Medidos	Observaciones
90% (224/255)	4200-4300 RPM	Máxima velocidad

Tabla 2 Mediciones de Velocidad

- **Observaciones:**
  - A bajos duty cycles (10%-30%), el motor presentaba un arranque suave pero con cierto ruido mecánico.
  - Entre 50%-70%, la respuesta fue casi lineal, ideal para control preciso.

8 Conclusiones

La práctica demostró que es posible medir con precisión la velocidad de un motor DC utilizando un encoder y el Arduino UNO, aprovechando las interrupciones y los temporizadores configurados en diferentes modos. El cálculo de RPM mediante pulsos del encoder resultó efectivo, aunque se observó que la precisión depende en gran medida de la resolución del encoder y de la eliminación de ruido eléctrico. El uso del L298N permitió un control adecuado del motor, aunque se recomienda implementar técnicas de filtrado para mejorar la estabilidad en bajas velocidades.

Durante la implementación, se identificaron desafíos como la configuración correcta de los temporizadores para evitar conflictos con funciones de Arduino (como millis() y analogWrite()), lo que requirió una gestión cuidadosa de los registros del microcontrolador. Además, se evidenció la importancia de optimizar el código para reducir el tiempo de ejecución de las interrupciones y evitar pérdida de pulsos del encoder. Esta práctica reforzó el entendimiento de la temporización en sistemas embebidos y destacó la necesidad de calibrar sensores para obtener mediciones confiables en aplicaciones de control de movimiento.

9 Referencias

[1] NXP Semiconductors, "I²C-bus specification and user manual," UM10204, Rev. 7.0, Oct. 2021. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[2] NXP Semiconductors, *"PCF8574 Remote 8-bit I/O expander for I²C-bus,"* Datasheet, Rev. 5, Sep. 2003. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/PCF8574.pdf

[3] Arduino, "Wire Library (I²C)," Arduino Reference. [Online]. Available: https://www.arduino.cc/en/reference/wire

[4] F. E. Rangel, "Comunicación entre microcontroladores: Protocolos I²C, SPI y UART," Revista Electrónica de Ingeniería, vol. 12, no. 2, pp. 45–60, 2020.

[5] J. Smith, "Digital Communication Systems: Practical Guide for Engineers," 3rd ed. New York, NY: Wiley, 2018, pp. 210–225.

[6] Texas Instruments, "Understanding the I²C Bus," Application Report SLVA704, Jun. 2015. [Online]. Available: https://www.ti.com/lit/an/slva704/slva704.pdf